DCC045 - Teoria dos Compiladores - Relatório Partes III - Análise Semântica

Grupo: Giovane Nilmer de Oliveira Santos - 201835012
Marcos Mateus Oliveira dos Santos - 201835019

Ajustes:

Especificação da Linguagem:

Foi reajustado no arquivo .g4 a especificação da linguagem da forma correta como foi especificado no documento. Essa definição incorreta faz com que a ordem de execução e o reconhecimento do programa tenham uma única ordem correta, que seria primeiro os tipos ***data***, seguidos das funções.

- Definição incorreta: `prog: data* fun*`
- Definição correta: `prog : (data | fun)*`

Visitor Expressão de Retorno de um função

Durante o desenvolvimento, foi identificado que o método ***visitFuncReturnExp*** estava gerando um nó nos casos em que não eram passados argumentos para uma função que possuía parâmetros definidos. Esse problema deveria ser validado no momento da análise semântica. O ajuste foi realizado para que a execução ocorresse de forma correta.

Código - `LangVisitor.java` :

```
public Node visitFuncReturnExp(LangParser.FuncReturnExpContext ctx) {
      int l = ctx.start.getLine();
      int c = ctx.start.getCharPositionInLine();
      String str = ctx.ID().getText();
      FunCallParams fCallPar = null;
      if (ctx.exps() != null) { // Adicionado nova verificação para
verificar ser o parametro é nulo
       fCallPar = (FunCallParams) ctx.exps().accept(this);
      }
      Exp exp = (Exp) ctx.exp().accept(this);
      return new FuncReturnExp(l, c, str, fCallPar, exp);
      }
```

Análise Semântica

Para a análise semântica, foi mantido o padrão Visitor para realizar a análise do programa, e o padrão Singleton foi utilizado nas classes para representar os tipos da linguagem. Foi empregada uma tabela de símbolos, onde cada função possui sua tabela local, que armazena as variáveis locais e os parâmetros da função. Ao processar um programa, os tipos **`data`** são executados, e cada função é percorrida para que suas tabelas locais sejam criadas.

Classes dos tipos:

Foram definidas 10 classes para representar os tipos primitivos da linguagem:

- **SType** -> Classe abstrata que representa todos os tipos da linguagem.
- **STyInt** -> Classe que representa o tipo primitivo inteiro.
- **STyNull** -> Classe que representa os valores nulos.
- **STyFloat** -> Classe que representa o tipo primitivo float.
- **STyChar** -> Classe que representa o tipo caractere.
- **STyBool** -> Classe que representa o tipo booleano.
- **STyData** -> Classe que representa os tipos ***data***.
- **STyArray** -> Classe que representa arrays de tipos ***data***.
- **FuncTypes** -> Classe que armazena os tipos das funções, os parâmetros, os tipos dos parâmetros e os tipos de retorno.

- **DataTypes** -> Classe que representa o tipo ***data***, armazenando os tipos e nomes de seus atributos.

Ambientes

Foi utilizada a definição de ambientes para armazenar um ambiente temporário para as funções, onde são armazenadas suas variáveis e os tipos associados. Além disso, foi criado um ambiente global que armazena os ambientes de cada função.

TyEnv<T>

```
public class TyEnv<T> {
	private TreeMap<String, T> tyEnv;
	public TyEnv() {
	tyEnv = new TreeMap<String, T>();
	}

	public void set(String id, T type) {
	this.tyEnv.put(id, type);
	}

	public T get(String id){
	return this.tyEnv.get(id);
	}


	public boolean checkDuplicity(String id) {
	return this.tyEnv.containsKey(id);
	}

	public String toString() {
		StringBuilder sb = new StringBuilder();
		sb.append("TyEnv: \n");
		for (String key : tyEnv.keySet()) {
		sb.append(key).append(" : ").append(tyEnv.get(key)).append("\n");
		}
		return sb.toString();
	}


}
```

TyEnv: Define um ambiente de tipos que possui um TreeMap, onde são associados os identificadores, como variáveis e funções, aos seus respectivos tipos. Foi atribuído um valor genérico para essa classe, permitindo assim armazenar qualquer tipo de dado.

Análise semântica e checagem de tipos

A classe principal para checagem de tipos e análise semântica é a ***LangVisitorTypeCheck***. Essa classe estende a implementação de ***LangVisitor*** e contém métodos para visitar cada tipo de nó da AST, realizando a análise semântica e a verificação de tipos.

Estrutura

```
private STyInt tyint = STyInt.newSTyInt(); // Cria um novo tipo inteiro
private STyFloat tyfloat = STyFloat.newSTyFloat(); // Cria um novo tipo float
private STyBool tybool = STyBool.newSTyBool(); // Cria um novo tipo booleano
private STyErr tyerr = STyErr.newSTyErr(); // Cria um novo tipo de erro
private STyChar tychar = STyChar.newSTyChar(); // Cria um novo tipo char
private STyNull tynull = STyNull.newSTyNull(); // Cria um novo tipo null
private HashMap<String, DataTypes> datas; // Armazena os tipos de datas
private ArrayList<String> logError; // Armazena os erros de tipo
private TyEnv<LocalEnv<SType>> env; // Armazena o ambiente de tipos
private LocalEnv<SType> temp; // Armazena o ambiente de tipos temporário
private Stack<SType> stk; // Pilha dos tipos
private boolean retChk; // Verifica o retorno da função
private int returnPos; // Posição do retorno
```

Lógicas para análise semântica e checagem de tipos

Atribuição (***AssignCommand***):

- **Atribuição de variáveis simples**: A expressão no lado direito é avaliada para verificar se seu tipo corresponde ao tipo da variável no lado esquerdo. Caso a variável já esteja presente no ambiente, seu tipo é verificado; caso contrário, a variável é adicionada ao ambiente. Se os tipos forem incompatíveis, é gerado um erro.
- **Atribuição de arrays**: O tipo do elemento no array é verificado para garantir que corresponde ao valor atribuído. Se o array não estiver presente no ambiente, ele é criado com o tipo correto. Se os tipos forem incompatíveis, é gerado um erro.
- **Atribuição a tipos *`data`***: O tipo do atributo do ***`data`*** é verificado. Se houver incompatibilidade entre os tipos, um erro é gerado.

Chamada de funções (*`FunctionCall`*):

- **Verificação da da função**: O identificador da função é obtido, e o ambiente de tipos ***LocalEnv<SType>*** da função é recuperado. Caso a função não esteja definida no ambiente, um erro é gerado.
- **Verificação do número de parâmetros**: São obtidos os parâmetros fornecidos na chamada da função e seus tipos esperados para os parâmetros da função O número de argumentos fornecidos é comparado com o número de parâmetros esperados. Se o número de parâmetros não for o mesmo, um erro é gerado.
- **Verificação dos tipos dos parâmetros**: Para cada parâmetro fornecido na chamada da função, o tipo da expressão é avaliado e comparado com o tipo esperado para o parâmetro correspondente. Se o tipo do argumento não coincidir com o tipo esperado, é gerado um erro.
- **Verificação dos valores de retorno**: Caso a função retorne valores, são comparados os valores retornados com os tipos definidos na função. Se os tipos ou o número de valores retornados não coincidir com o esperado:

Acesso a arrays (*`ArrayAccess`*):

- **Acesso a arrays simples**:
  - Se o valor acessado for um identificador variável, é verificado se o array está definido no ambiente.
  - Se o array não estiver definido, é gerado um erro.
  - Se estiver definido, é verificado se a variável é realmente um array. Se for, o tipo do array é empilhado para uso posterior, caso contrário, um erro é gerado.
- **Acesso a matrizes (arrays de arrays)**:
  - Se o valor acessado for outro ***`ArrayAccess`*** (matriz), é verificado se o array está definido. Se não estiver, gera um erro.
  - Se for um array válido, é verificado se é uma matriz.
  - É gerado um erro é gerado caso seja uma matriz de três dimensões .
  - O índice do array é verificado para garantir que seja um número inteiro. Se o índice não for inteiro, um erro é gerado.
- **Empilhamento dos tipos**:
  - O tipo do array ou do elemento acessado é empilhado para ser utilizado nas verificações posteriores.

Tipando funções:

retChk: é usada para verificar se a função possui uma instrução ***return***. O ambiente temporário (temp) da função é configurado com os parâmetros e comandos da função.

Verificação dos parâmetros:

- Se a função possui parâmetros, são processados e seus tipos são armazenados no ambiente das variáveis temp, associando os identificadores dos parâmetros com seus respectivos tipos.

Verificação dos comandos:

- Cada comando dentro da função é visitado e processado.
- Durante essa verificação, é verificado se o último comando é um IfCommand. Se for, há uma verificação extra para garantir que a função tenha um return no final do escopo.

Verificação do return:

- Após percorrer os comandos, é verificado se a função deve retornar um valor, com base nos tipos de retorno definidos.
- Se não houver um return no final da função e ele for necessário, são gerados erros:
    - Se a função deveria ter um return mas não tem.
    - Se a função está correta, mas faltou o return no final de um bloco condicional, outro tipo de erro é gerado.

Bateria de testes:

Testes semanticamente incorretos: Todos os testes semanticamente incorretos geraram erros de validação detectados pela análise semântica.

```
Testando testes/semantica/errado/instanciate.lan
[ (8, 19)- l variável indefinida
  (9, 3) Atribuição inválida: tipos diferentes para z
FALHOU]
```

```
Testando testes/semantica/errado/data0.lan
[ (8, 4) Atribuição inválida: tipos diferentes, Int atribuído a x:Float
FALHOU]
Testando testes/semantica/errado/if_oneCMD.lan
[ (4, 5) Atribuição inválida: tipo diferente para x
FALHOU]

Testando testes/semantica/errado/function_call_expr.lan
[ (9, 6) Tipo inválido para expressão: Int e Float
FALHOU]

Testando testes/semantica/errado/parameters.lan
[ (2, 0) Parâmetros duplicados: x : Int, x : Float
  (3, 7) Tipo inválido para expressão: Int e Float
  (9, 6) Tipo inválido para expressão: Int e Float
FALHOU]

Testando testes/semantica/errado/data1.lan
[ (8, 4) Atributo z não encontrado no tipo x
FALHOU]

Testando testes/semantica/errado/function_call_ret_use.lan
[ (9, 2) variável z tipo: Int diferente do retornado pela função Float
FALHOU]

Testando testes/semantica/errado/instanciate1.lan
[ (8, 19)- l variável indefinida
  (9, 3) Atribuição inválida: tipos diferentes para z
FALHOU]

Testando testes/semantica/errado/ifelse_oneCMD.lan
[ (2, 5)- x variável indefinida
FALHOU]

Testando testes/semantica/errado/main_args.lan
[ (2, 0) A função main não deve possuir argumentos
  (9, 2) Quantidade de retornos solicitados diferente do número definido na
função
FALHOU]

Testando testes/semantica/errado/parameters2.lan
[ (9, 8) Quantidade de argumentos incompatível com o definido na função f
FALHOU]
```

```
Testando testes/semantica/errado/function0.lan
[ (7, 2) Número de argumentos fornecidos não corresponde ao número de
parâmetros esperados
FALHOU]

Testando testes/semantica/errado/if_teste.lan
[ (3, 2) O tipo da expressão do if deve ser bool
FALHOU]

Testando testes/semantica/errado/attrAND.lan
[ (2, 6) Tipo inválido para expressão de comparação &&
FALHOU]

Testando testes/semantica/errado/parameters1.lan
[ (9, 8) Foram enviados argumentos para uma função onde não foram definidos
parâmetros
FALHOU]

Testando testes/semantica/errado/data3.lan
[ (2, 11) O tipo data Ponto não foi definido para ser um parâmetro
  (3, 9) x tipo data p indefinido
FALHOU]

Testando testes/semantica/errado/teste8.lan
[ (22, 7) Quantidade de argumentos incompatível com o definido na função
spook
FALHOU]

Testando testes/semantica/errado/function1.lan
[ (7, 2) Quantidade de retornos solicitados diferente do número definido na
função
FALHOU]

Testando testes/semantica/errado/main_missing.lan
[ (2, 0) Função main não definida
FALHOU]

Testando testes/semantica/errado/return.lan
[ (2, 0) Falta do return no final do escopo da função f
  (10, 6) Tipo inválido para expressão: Int e Float
FALHOU]
```

```
Testando testes/semantica/errado/attrFALSE.cmd
[ (2, 6) flaes variável indefinida
FALHOU]

Testando testes/semantica/errado/data2.lan
[ (6, 0) O tipo data Ponto já foi definido
FALHOU]

Total de acertos: 0
Total de erros: 22
```

Testes semanticamente corretos: Todos os testes semanticamente corretos foram executados com sucesso.

```
Testando testes/semantica/certo/teste1eMeio.lan            [  OK  ]
Testando testes/semantica/certo/teste1.lan                 [  OK  ]
Testando testes/semantica/certo/teste3.lan                 [  OK  ]
Testando testes/semantica/certo/teste7.lan                 [  OK  ]
Testando testes/semantica/certo/teste2.lan                 [  OK  ]
Testando testes/semantica/certo/teste6.lan                 [  OK  ]
Testando testes/semantica/certo/teste8.lan                 [  OK  ]
Testando testes/semantica/certo/teste9.lan                 [  OK  ]
Testando testes/semantica/certo/teste0.lan                 [  OK  ]
Testando testes/semantica/certo/teste12.lan                [  OK  ]
Testando testes/semantica/certo/teste11.lan                [  OK  ]
Testando testes/semantica/certo/teste5.lan                 [  OK  ]
Testando testes/semantica/certo/teste4.lan                  [ OK  ]
Total de acertos: 13
Total de erros: 0
```

Estrutura de Diretórios, Compilação e Execução

Estrutura de Diretórios

1. `lib/`: Contém bibliotecas necessárias, incluindo o ANTLR4.
2. `lang/`: Contém o código-fonte do compilador, incluindo os diretórios `parser`, `interpreter`, e `ast`.
3. `parser/`: Código gerado pelo ANTLR e classes de parser.
4. `interpreter/`: Classes responsáveis pela interpretação dos códigos.
5. `ast/`: Classes que representam os nós da Árvore de Sintaxe Abstrata.
6. `semantic/`: Classes responsáveis pela análise semântica. (Nova)

Compilação

O projeto usa um Makefile para simplificar o processo de construção. Os seguintes comandos estão disponíveis:

Compilar Todo o Projeto

```
make all
```

Limpa caches antigos, gera o parser a partir do arquivo `.g4`, compila a AST, o parser, o interpretador e o código principal.

Limpar Cache

```
make clean-cache
```

Limpa arquivos de classes e caches gerados anteriormente.

Gerar o Parser com ANTLR

```
make gen-parser-antlr
```

Gera o código do parser a partir da definição da gramática no arquivo .g4 (Lang.g4).

Compilar o Parser

```
make gen-parser
```

Compilar o Semantic (Novo)

```
Make gen-semantic
```

Compilar o Interpretador

```
make gen-interpreter
```

Compilar a Árvore de Sintaxe Abstrata

```
make gen-ast
```

Execução

Para executar o compilador e interpretar um arquivo específico, utilize:

```
java -cp $(ANTLR_JAR):. lang.LangCompiler -i <caminho_para_arquivo>
```

Onde `<caminho_para_arquivo>` deve ser substituído pelo caminho do arquivo Lang que deseja compilar e executar.

Testes

Para executar testes automatizados sobre o parser, analisador semântico ou interpretador:

```
make run-test-semantic (NOVO)
make run-test-parser
make run-test-it
```